

Testiranje softvera

U okviru ove lekcije pružićemo uvid u procese na kojima se temelji testiranje programa, planove koje je potrebno realizovati i ishode planiranja koje je potrebno postići da bi se dobili kvalitetni testovi, a samim tim i pouzdani rezultati tog testiranja u svrhu obezbeđenja kvaliteta.

Već par puta smo spomenuli skup pravila softverskog razvoja – SWEBOK i samu oblast softverskog inženjerstva. Na osnovu spomenutih, možemo sagledati 10 oblasti znanja na polju softverskog inženjerstva:

1. Software Requirements
2. Software Design
3. Software Construction
4. Software Testing
5. Software Maintenance
6. Software Configuration Management
7. Software Engineering Management
8. Software Engineering Process
9. Software Engineering Tools And Methods
10. Software Quality



Slika 3.1. Životni ciklus razvoja softvera

Sve ove oblasti direktno formiraju životni ciklus razvoja softvera i, kao što možete videti softversko testiranje i obezbeđenje kvaliteta pripadaju životnom ciklusu; iako se QA ne prikazuje kao odvojena faza životnog ciklusa, ona pripada fazi testiranja. Sada ćemo odgovoriti na nekoliko pitanja koja su jako bitna za razumevanje na samom početku ove priče o testiranju softvera.

Zašto sprovodimo testiranje?

Prvo i osnovno pitanje bi bilo: „Zašto sprovodimo testiranje?“ Na ovo pitanje smo dali odgovor tokom prethodnih lekcija – govorili smo koji se troškovi javljaju kada imamo program sa defektima, koji to uticaj ima na našu kompaniju i slično. Sada ćemo samo ukratko, po tezama, da sumiramo zašto je testiranje nužan proces u razvoju programa:

- Zato što želimo da otklonimo što više defekata u softveru pre isporuke krajnjim korisnicima. Obratite pažnju na to da govorimo *što više defekata*, jer nikada se mogu ukloniti baš sve greške, jer je za neke potrebno da softver krene sa realizacijom i tek tada možete otkriti neke probleme koje nismo mogli otkriti tokom razvoja.
- Akcenat kod testiranja je na tome da se otkrije što je moguće više defekata koji su teški sa materijalnog ili bezbednosnog aspekta. Dakle, tamo gde greška softvera može ugroziti život i zdravlje radnika, npr. kod softvera za upravljanje hemijskim procesima, velikim mašinama i slično. Zabeležen je veliki broj slučajeva gde su kompanije gubile velike količine novca usled grešaka programa ili usled hakerskih upada, gde su ukradene lične informacije registrovanih korisnika.

Da li testiranje ima uticaj na kod i njegovu strukturu?

Aktivnost testiranja softvera se nekada smatra destruktivnom prema programskom kodu. Ono će, naravno, imati uticaj na kod, jer ćemo, kako otkrivamo greške, unositi izmene i vršiti dopune koda i čitav update projekta. Sa druge strane, iako je reč o destruktivnom procesu jer menja naše inicijalne pretpostavke kako kod napisati, na kraju se pokazuje da je ta aktivnost ipak konstruktivna, jer čini naš kod otpornijim na greške.

Sa aspekta strukture koda, rezultat aktivnosti testiranja softvera je programski kod visoke pouzdanosti i velike otpornosti (robustan), stabilan, kao i potvrda da softver zadovoljava zahteve krajnjeg korisnika.

Važan aspekt koda i njegove strukture jeste i to da, ukoliko testiranje vršimo za drugo lice – dakle, kada softver nije naš i mi nećemo direktno vršiti uticaj na kod, pišemo dokumentaciju o tome šta nije bilo u redu, gde je tačno problem, zašto je to problem itd. Dakle, ne vršimo ispravke i dopune koda, već samo vodimo beleške – koje greške je program generisao i kako to pravi problem u radu programa.

Zašto sa testiranjem treba početi u ranim fazama razvoja?

Proces razvoja softverskog proizvoda (slika 3.1) počinje specifikacijom zahteva, a zatim slede faze koje su sve složenije i uključuju sve više ljudi u razvoj, dok cena projekta sve više raste. Greška otkrivena na kraju takvog procesa zahteva mnogo više napora da se ispravi. U ove svrhe se piše posebna dokumentacija, pod nazivom *Test slučajevi*; oni se uglavnom pišu odmah posle usvajanja zahteva i dostave razvojnom timu, radi boljeg sagledavanja traženog cilja.

Kada problem naraste, tada je softverska greška veća. Najbolje je eliminisati probleme u ranijom fazama. Postoji čitav pristup razvoju softvera koji se zove *test driven development*, gde je fokus na pisanju koda na način da ga je lako testirati. *Test driven developmentu* posvetićemo i deo ovoga kursa.

Zašto se javljaju greške?

Postoji veliki broj razloga zašto se greške javljaju. One, naravno, zavise i od tipa projekata na kojima radimo, ali generalno, greške nastaju iz sledećih razloga:

- Nepoštovanje koraka softverskog inženjerstva
 - Greške usled primene pogrešne metodologije, preskakanja faza životnog ciklusa softvera itd.
- Nepotpuna ili pogrešna specifikacija zahteva
 - Npr. UserID podesiti tako da ne postoje dva korisnika sa istim korisničkim ID-jem.
Ovo je jedan od zahteva softvera, ali nije dovoljno precizno definisan, jer ostaje puno otvorenih pitanja. Da li UserID može da bude negativna vrednost? Da li je u igri ceo broj ili broj sa pokretnim zarezom?...
 - U ovom slučaju treba precizno definisati zahtev, npr.:
UserID definisati kao pozitivan integer maksimalne duže 4 bajta i inkrementovati ga da bismo bili sigurni da ne postoje dva korisnika sa istim vrednostima ID-ja.
- Kupci u fazi planiranja ne znaju šta im je potrebno
 - Primera radi, klijent želi da u svom softveru pruži korisnicima opciju da kreiraju nalog u kojem mogu da unesu neke osnovne informacije o sebi. U toku procesa razvoja, klijent odlučuje da želi da korisnici imaju opciju dodavanja svoje slike, koja će se potom prikazivati na svim mestima gde korisnik obavlja unos podataka. Ovo je drastično izmenjen zahtev od inicijalnog, što pravi problem razvojnom timu, koji je primoran da menja i dodaje programsku logiku, što na kraju može izazvati niz grešaka u radu.
- Nedovoljno poznavanje domena primene
 - Razvojni tim nije dovoljno upoznat sa delatnostima klijenta, poput izrade softvera u poljoprivredi, gde opet morate imati makar osnovno razumevanje procesa rada da možete predvideti kakve greške mogu nastati i znati da preciznije definišete neki zahtev programa.
- Promena tehnologije
 - Promena tehnologije je jedan od najčešćih razloga pojave grešaka. U ovom slučaju greške nastaju ukoliko menjamo verziju programskog jezika, verziju biblioteka ili celokupne biblioteke koje koristimo za projekat.

Pored navedenih, mogu postojati i drugi podjednako važni razlozi pojave grešaka, poput:

- složenosti poslovnih procesa;
- nedostatka znanja i iskustva razvojnog tima;
- nerazumevanja i nedovoljno komunikacije između članova tima;
- nepažnje, nedostatka koncentracije, umora;
- pritiska zbog rokova.

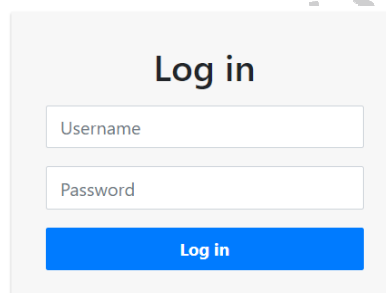
Sada smo izvršili pregled zašto je testiranje potrebno, o čemu voditi računa prilikom planiranja razvoja softvera i kako greške mogu nastati, i sada dolazimo do toga kako testiranje započinjemo. Krenućemo od prvog, osnovnog dela svakog testiranja, a to je test slučaj.

Test slučaj

Test slučaj predstavlja skup akcija koje je potrebno izvršiti da bismo verificovali pravilan rad određene funkcije ili funkcionalnosti našeg softvera. Svaki test slučaj sadrži precizno definisane korake, podatke koje koristimo, ishode koji treba da budu ispunjeni na početku i ishode koji treba da budu ispunjeni nakon testiranja. Razvojni tim na kraju upoređuje sve podatke da bi se potvrdio pravilan rad dela programa, kao i njegova usaglašenost sa očekivanjima klijenta.

Sada vam ovo možda deluje apstraktno, ali verujemo da ste u dosadašnjem radu sigurno radili po prethodno navedenom principu. Primera radi, ako ste pisali program koji omogućava unos podataka, sigurno ste proveravali da li unos radi pravilno, da li možete da kliknete na polje, da li možete da upišete podatke u njega i da li se podaci negde upisuju. Ovo je upravo nešto što se zove test scenario, gde znate koje je korake potrebno uraditi da biste se uverili da nešto zaista radi kao što je očekivano. Test slučaj je vrlo sličan, samo što je preciznost prilikom provere veća. Pa pogledajmo jedan uprošćen primer, gde možemo lako uočiti kako se razvija test slučaj i kako se proverava jedna funkcija u okviru softvera.

Recimo da želimo da proverimo jednu jednostavnu login formu u našem programu, poput forme prikazane na slici u nastavku:



Slika 3.2. Primer jednostavne login forme

Test slučajevi za testiranje forme bi mogli biti:

Test Slučaj 1:

Proveriti rezultat nakon unosa validnog korisničkog imena i lozinke

Test Slučaj 2:

Proveriti rezultat nakon unosa pogrešnog korisničkog imena i lozinke

Test Slučaj 3:

Proveriti odgovor kada je polje za unos korisničkog imena prazno i kada se klikne Login dugme.

Naravno, možemo napisati još više slučajeva kada proveravamo: jedno polje popunjeno, a drugo prazno; pogrešni karakteri na unosu i slično, ali ostaćemo sada na ova tri slučaja i proširiti svaki od njih.

Ovo je bio prvi korak pisanja test slučaja, gde smo definisali njegov opis. Sledeći korak je da definišemo i tačne podatke koje ćemo koristiti za test. Beleženje podataka koje smo koristili je važno, jer je bitno da možemo da ponovimo pojavu greške. Tada znamo da nešto u tom određenom testu otkriva grešku, a ne da je ona nastala nezavisno od testa. Definisanje podataka za test može izgledati ovako:

Redni broj test slučaja	Opis test slučaja	Podaci korišćeni za test
1	Proveriti rezultat nakon unosa validnog korisničkog imena i lozinke	Username: <u>adminemail@gmail.com</u> Password: admin123
2	Proveriti rezultat nakon unosa pogrešnog korisničkog imena i lozinke	Username: <u>wrongemail@gmail.com</u> Password: wrong123
3	Proveriti odgovor kada je polje za unos korisničkog imena prazno i kada se klikne Login dugme	Username: Password: admin123

Tabela 3.1. Primer korišćenih podataka za svaki od test slučajeva

Dakle, imamo definisane podatke za svaki od slučajeva. Sledeći korak je da precizno definišemo korake koje obavljamo u okviru testa slučaja. Naravno, za ovaj primer će oni biti jednostavni, poput unosa email adrese, ali ovaj princip se prati kako za male, jednostavne funkcije tako i za velike i kompleksne. Pa, pogledajmo kako bi to izgledalo na našem primeru:

Redni broj test slučaja	Opis test slučaja	Koraci testa	Podaci korišćeni za test
1	Proveriti rezultat nakon unosa validnog korisničkog imena i lozinke	1) Uneti korisničko ime 2) Uneti lozinku 3) Kliknuti Login dugme	Username: <u>adminemail@gmail.com</u> Password: admin123
2	Proveriti rezultat nakon unosa pogrešnog korisničkog imena i lozinke	1) Uneti pogrešno korisničko ime 2) Uneti pogrešnu lozinku 3) Kliknuti Login dugme	Username: <u>wrongemail@gmail.com</u> Password: wrong123
3	Proveriti odgovor kada je polje za unos korisničkog imena prazno i kada se klikne Login dugme	1) Polje za unos korisničkog imena ostaviti prazno 2) Uneti pravilnu lozinku	Username: Password: admin123

		3) Kliknuti Login dugme	
--	--	-------------------------	--

Tabela 3.2. Primer koraka izvršavanja testa za svaki od test slučajeva

Sledeći korak je definisanje očekivanog rezultata nakon izvršenja test slučaja. Dakle, u ovom delu navodimo šta treba da se dogodi. Recimo, ako je uneta pogrešna lozinka, očekivani rezultat bi bio da login nije uspešan. Pogledajmo ovaj korak:

Redni broj	Opis test slučaja	Koraci testa	Podaci korišćeni za test	Očekivani ishod
1	Proveriti rezultat nakon unosa validnog korisničkog imena i lozinke	1) Uneti korisničko ime 2) Uneti lozinku 3) Kliknuti Login dugme	Username: <u>adminemail@gmail.com</u> Password: admin123	Login treba da bude uspešan
2	Proveriti rezultat nakon unosa pogrešnog korisničkog imena i lozinke	1) Uneti pogrešno korisničko ime 2) Uneti pogrešnu lozinku 3) Kliknuti Login dugme	Username: <u>wrongemail@gmail.com</u> Password: wrong123	Login ne sme biti uspešan
3	Proveriti odgovor kada je polje za unos korisničkog imena prazno i kada se klikne Login dugme	1) Polje za unos korisničkog imena ostaviti prazno 2) Uneti pravilnu lozinku 3) Kliknuti Login dugme	Username: Password: admin123	Login ne sme biti uspešan

Tabela 3.3. Primer unetih očekivanih ishoda

Za kraj pisanja test slučajeva navodimo šta se dogodilo nakon izvršenja testa slučaja i u posebnoj koloni označavamo da li je test uspešno prošao ili ne. Ukoliko test nije prošao, to znači da je došlo do greške i neočekivanog ponašanja.

Na primeru jednog od naših test slučajeva, to bi izgledalo ovako:

Redni broj	Opis test slučaja	Koraci testa	Podaci korišćeni za test	Očekivani ishod	Ishod testa	Uspešan test (DA/NE)
1	Proveriti rezultat nakon unosa validnog korisničkog imena i lozinke	1) Uneti korisničko ime 2) Uneti lozinku 3) Kliknuti Login dugme	Username : <u>adminem</u> <u>ail@gmail</u> <u>.com</u> Password: admin123	Login treba da bude uspešan	Login uspešan	DA

Tabela 3.4. Izveden test slučaj

Dakle, ovaj postupak se ponavlja za sve slučajeve testiranja. Naravno, kako je softversko testiranje već razvijena oblast, tako imamo i alate koji nam olakšavaju rad sa testovima slučaja i automatizuju proces. Jednu od platformi ovog tipa možete videti na sledećem linku:

<https://qase.io/>

Prikazani test slučaj je definisan i kroz neke osnovne principe testiranja softvera kojima se generalno vodimo kada kreiramo test slučajeve i generalno testiramo softver:

1. Obavezan deo test slučaja je i navođenje očekivanog izlaza ili rezultata.
2. Programer treba da izbegava testiranje sopstvenog programa, jer će teže uočiti greške.
3. Programerska organizacija ne bi trebalo da testira sopstvene programe, iz istog razloga.
4. Svaki proces testiranja treba da uključi detaljnu inspekciju rezultata svakog testa, jer problem može postojati i u samom testu.
5. Test slučaj mora biti napisan za ulaze koji su nevalidni i neočekivani, kao i za one koji su validni i očekivani.
6. Ispitivanje programa radi provere da li ne radi ono što treba da radi je samo polovina posla; druga polovina je provera da li radi ono što ne bi trebao da radi. U slučaju našeg primera, program ne bi smeo da ulogu korisnika ako on nije uneo korisničko ime, već da obavesti o neispravno unetim podacima.
7. Ne pristupati testiranju sa pretpostavkom da greške neće biti pronađene.

Pored testa slučaja, postoji i odvojena dokumentacija, koja se najčešće definiše na nivou projekt menadžera ili team leadera; to je test plan. Test plan je dokument u kojem je opisana organizacija procesa testiranja. Ovim dokumentom opisujemo opseg, pristup, resurse i raspored planiranih aktivnosti testiranja. Kompleksni sistemi mogu imati i test plan visokog nivoa, koji obrađuje sveobuhvatne zahteve celog sistema, i posebne test planove za podsisteme i komponente sistema. Nekada se uvode test planovi za svaki nivo testiranja, pa

tako postoji test plan jedinice, test plan integracije i sistemski test plan. U ovom planu se često navodi kompletna lista funkcionalnosti koje podležu testiranju. Kao što smo i ranije naveli, ovo se obavlja na višim organizacionim nivoima i prema standardima organizacije, te ne postoji striktno određen sadržaj planova dokumenta, ali oni koje zanima više na ovom linku mogu preuzeti jedan primer generičkog test plana, gde se mogu videti sve sekcije i šta se očekuje da se u okviru njih nalazi.

Tehnike testiranja programa

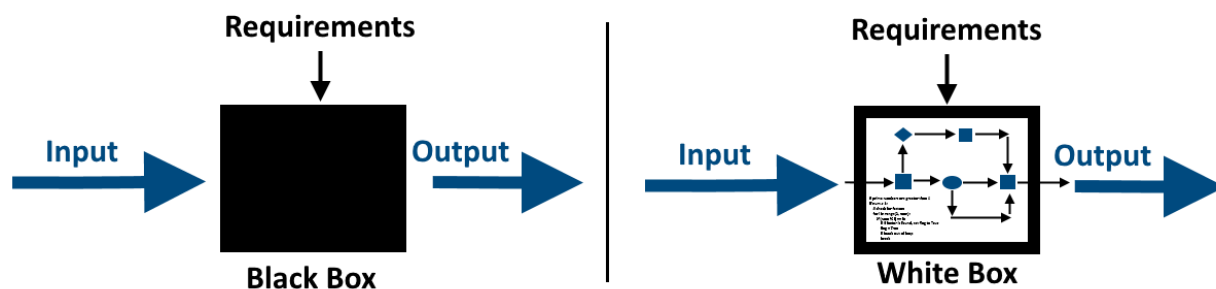
Metodologije softverskog testiranja su stvorile veliki broj strategija i tehnika koje se koriste u svrhu provere ponašanja programa. Tehnike testiranja obuhvataju provere svega od front do backend testiranja, preko testiranja malih celina, do testiranja celog sistema softvera. U okviru ove lekcije, dodirnućemo površinu tehnika testiranja, a ostatak kursa ćemo posvetiti upravo konkretnom radu na programima sa upotrebom tehnika koje ćemo navesti u nastavku.

Testiranje se u osnovi može podeliti na dva načina: prema pristupu sistemu i prema nivou testiranja. Prema pristupu sistemu, testiranje se deli na:

1. Funkcionalno testiranje – Ovaj vid testiranja je baziran na specifikaciji.

Funkcionalno testiranje je tip softverskog testiranja kojim se vrši validacija softvera u odnosu na definisane zahteve i specifikaciju. Svrha ovog vida testiranja je da se testira svaka funkcionalnost programa tako što vršimo neke interakcija ili unose podataka u program, a proveravamo samo da li na izlazu dobijamo očekivani podatak ili promenu u interfejsu. Funkcionalno testiranje se prvenstveno oslanja na black box metod testiranja, gde nas ne zanima izvorni kod programa, već samo da li program obavlja očekivanu radnju.

Testovi su nezavisno od konkretne implementacije, tako da su upotrebljivi i ukoliko dođe do promene implementacija, dok razvoj testova može teći paralelno sa razvojem programa.



Slika 3.3. Black box i white box testiranje

2. Strukturno testiranje – Kod ovog vida testiranja, testiranje je bazirano na samom kodu.

Strukturno testiranje se fokusira na samu implementaciju programa i dostupni kod. Naziva se još metoda *white box*, nekada i *glass box*. Fokus je na izvršavanju svih programskih struktura i struktura podataka u softveru koji se testira i na osnovu toga se određuju testovi. U ovom tipu testiranja, ne proverava se specifikacija, pa samim tim nije moguće ni otkriti da li su sve zahtevane funkcionalnosti zaista implementirane u program, već se vrši proverava izraza, uslova, putanja.

Da sumiramo, karakteristike black box testiranja su:

- tehnike dizajnirane bez poznavanja interne strukture programa i dizajna;
- zasnovano je na funkcionalnim zahtevima dogovorenim u fazi planiranja projekta;
- poznato je šta softver treba da radi;
- često se naziva i funkcionalno testiranje.

Dok su karakteristike white box testiranja:

- ispitivanje internog dizajna programa;
- zahteva detaljno poznavanje strukture i uvid u programski kod;
- često se naziva i strukturalno testiranje.

Testiranje se takođe može deliti prema nivou; tada podela izgleda ovako:

Testiranje jedinica

Testiranje jedinica (unit testing) odnosi se na testiranje pojedinačnih jedinica izvornog koda ili delova klase. Najmanja funkcionalna jedinica koda je najčešće jedna metoda unutar klase, ali, se, naravno, mogu testirati i druge celine poput rutina, funkcija, klasa i drugih. Testiranje jedinica najčešće koriste sami programeri, kako bi testirali svoj napisani kod. Testiranje jedinica možemo posmatrati kao pisanje malo koda kojim ćemo proveriti neki drugi kod. Kod testova jedinica testiranje je najčešće potpuno automatizovano, tj. programer jednom piše test, a test se zatim izvršava više puta i onoliko koliko je potrebno.

Integraciono testiranje

Nakon izvršenog testiranja jedinica, ispravne jedinice se integrišu u celine poput paketa i modula. Tada na scenu stupa integraciono testiranje; glavni fokus u ovom vidu testiranja je na verifikaciji funkcionalnosti i veza (interfejsa) između integrisanih modula.

Testiranje sistema

Na kraju, kada su svi delovi završeni i provereni, započinje se sa testiranjem sistema, gde proveravamo ponašanje sistema kao celine u odnosu na očekivane zahteve koje sistem treba da ispuni. Kako je većina funkcionalnih zahteva proverena u okviru nižih nivoa testova, ovde se akcenat postavlja na nefunkcionalne zahteve, poput brzine, pouzdanosti, robusnosti, sigurnosti itd.

U okviru ove lekcije predstavili smo oblast testiranja, govorili smo o načinima kako se realizuje testiranje i sa kojom svrhom. Na samom kraju prikazali smo tehnike i vrste testova koji se generalno koriste, a u narednim lekcijama zapoćemo i sa konkretnim radom i pisanjem ovih testova u okviru Pythona.

Pitanje

Označite tačan odgovor:

Black box testiranje se često naziva i:

- integraciono testiranje
- **funkcionalno testiranje**
- strukturalno testiranje

Objašnjenje:

Black box testiranje se često naziva i funkcionalno testiranje.

Rezime

- Aktivnost testiranja softvera se nekada smatra destruktivnom prema programskom kodu, jer u slučaju otkrivanja grešaka unosimo izmene i dopune koda. Na kraju razvojnog procesa se testiranje ipak pokazuje kao konstruktivna aktivnost, jer čini naš kod otpornijim na buduće greške.
- Sa testiranjem treba početi u ranim fazama razvoja, jer kako napredujemo u fazama, one postaju sve složenije, uključuju sve više ljudi u razvoj i cena projekta sve više raste. Greška otkrivena na kraju takvog procesa zahteva mnogo više napora da se ispravi.
- Test slučaj predstavlja skup akcija koje je potrebno izvršiti da bismo verifikovali pravilan rad određene funkcije ili funkcionalnosti našeg softvera.
- Kod funkcionalnog testiranja, testiranje vršimo kroz interakciju sa interfejsom programa ili unosimo podatke na ulaz programa. Nakon toga samo proveravamo da li na izlazu dobijamo očekivani podatak ili promenu u interfejsu. Funkcionalno testiranje predstavlja black box tehniku testiranja.
- Strukturalno testiranje predstavlja testiranje koje je bazirano na samom kodu, gde je fokus na samoj implementaciji programa i dostupnom kodu. Naziva se još metodom white box, nekada i glass box.
- Testiranje se takođe možete deliti prema nivou, na: testiranje jedinica, integraciono testiranje i sistemsko testiranje.
- Testiranje jedinica (unit testing) odnosi se na testiranje najmanjih funkcionalnih delova koda. Jedinica je najčešće jedna metoda unutar klase, ali se, naravno, mogu testirati i druge celine poput rutina, funkcija, klasa i drugih. Testiranje jedinica najčešće koriste sami programeri, kako bi testirali svoj napisani kod.
- Integraciono testiranje se obavlja nakon izvršenog testiranja jedinica – ispravne jedinice se integrišu u celine poput paketa i modula, gde se nakon povezivanja ovih paketa testira funkcionalnost veza između integrisanih modula.
- Testiranje sistema se obavlja na samom kraju, kada su svi delovi završeni i provereni. Tada ostaje da proverimo ponašanje sistema kao celine u odnosu na očekivane zahteve koje sistem treba da ispuni. Kod ovoga vida testiranja akcenat je na nefunkcionalnim zahtevima, poput brzine, pouzdanosti, robusnosti, sigurnosti itd.